Lund University
Ericsson Mobile Platforms

Master Thesis

Camera focus controlled by face detection on GPU

Authors:
Karl Berggren
Master of Science in Computer Science and Engineering
Lund University
Pär Gregersson
Master of Science in Computer Science and Engineering
Lund University

Advisors:
Beatriz Grafulla
New Technology Research Department
Ericsson Mobile Platforms
Jon Hasselgren
Department of Computer Science
Lund University

Abstract

This master thesis investigates the possibility to use the GPU of a mobile platform to detect faces and adjust the focus of the camera accordingly. Requirements for such an application are given, and different methods for face detection are described and evaluated according to these requirements. A GPU implementation of the cascade classifier is described. A CPU reference implementation with results is also presented. Methods for setting focus correctly when faces are detected are discussed. Issues when working with OpenGL ES 2.0 are described, and finally future work in the area is proposed.

October 16, 2008

Contents

Chapter 1

Introduction

The introduction of graphics processing units in mobile devices have spawned a new field of research. Developers are not only interested in the use of 3D graphics, but also in the possibility to use this new hardware for other purposes. Concurrently many mobile devices today are equipped with cameras. These are rapidly evolving, merging the fields of compact digital cameras and mobile phones. When adjustable focus capabilities became available on mobile devices, the demand for intelligent focus control arose. Compact digital cameras are often equipped with face detection as part of the focus control, a similar feature is desirable in mobile devices such as mobile phones. Both hardware and software implementations of such face detection exist today.

The purpose of this master thesis is to examine if graphics hardware can be used to perform face detection in order to control the focus of a camera module. This thesis will hopefully help to determine if face detection algoritms need to be integrated in the camera ISP block, or if they can be implemented using graphics hardware. The conclusions of this thesis might also be of interest to GPGPU developers in other fields than face or object detection.

The work plan was to study existing face detection algorithms, in order to find one suitable for GPU implementation. We were supposed to get familiar with the supplied platform and set up an image flow from the camera to the screen. The selected face detection algorithm should be used to detect faces and adjust the focus of the camera accordingly.

Note that this report is primarily directed to software developers and researchers with prior knowledge of the OpenGL API and graphics hardware programming.

1.1 Terminology

GRAPHICS RELATED	
GPU	Graphics processing unit.
GPGPU	General-purpose computing on graphics processing units.
OpenGL	Open Graphics Library, a cross-language and cross-platform API for writing applications that uses 2D and 3D graphics.
OpenGL ES	OpenGL for embedded systems.
GLSL	OpenGL Shading Language, a language specification for programming graphics hardware.
GLSL ES	OpenGL Shading Language for embedded systems.

FACE DETECTION	
Detection window	The region of an image where the face detector is currently tested. It works as a sliding window in the image that is scanned.
Face	An image or region of an image containing a human face.
Non-face	An image or region of an image not containing a human face.
Correct detection	An image or region of an image correctly classified as a face.
False Positive	An image or region of an image incorrectly classified as a face.
Feature	A property in an image, ex. a feature can be that a certain region is darker than the region surrounding it.
Weak classifier	A weak classifier can use one or a few features together with thresholds to classify an image as a face or non-face.
Strong classifier	A combination of weak classifiers with corresponding weights used to determine whether an image or a region of an image contains a face.
Cascade classifier	A sequence of strong classifiers that classifies an image or a region as a face or non-face, by discarding non-faces as early as possible in the sequence.

HARDWARE AND IMPLEMENTATION	
DirectFB	Direct Frame Buffer is a thin library that provides hardware graphics acceleration, input device handling and abstraction on top of the Linux Framebuffer Device.
ISP	Image signal processing.
V4L2 - Video For Linux 2	This is a video capture/overlay API of the linux kernel.

1.2 Working Environment

For this master thesis project, we were given access to an ARM based platform with OpenGL ES 2.0 acceleration and a QVGA touch screen emulating a mobile device. The platform had a single core Universal Scalable Shader Engine for both vertex and fragment program execution. The hardware platform also had a 5 megapixel camera with autofocus and camera drivers implementing Video For Linux 2.

1.3 Thesis outline

This thesis is structured as follows:

- In chapter 2, we discuss the requirements for the face detector, and evaluate some detection methods according to these requirements. We also provide details of the selected method.
- In chapter 3, we explain how the face detector was constructed according to the selected method.
- In chapter 4, we give a suggestion of how the face detector can be implemented on GPU, and how a reference implementation running on the CPU of the supplied platform was implemented.
- In chapter 5, we describe the complete setup of the detection on the platform. Fetching images, performing detection, presenting the result on screen and handling the focus.
- Lastly, in chapter 6, we draw conclusions from our results, discuss issues we encountered and suggest future work.

Chapter 2

Face detection methods

Face detection is an extensive research field, where numerous methods have been investigated. When constructing a face detector, its intended use has to be considered, since a universal detector is not the best solution for every situation. The correct detection and the false positive rates are the two most important parameters. An ideal face detector would yield a correct detection rate of 1, and a false positive rate of 0. In practice however, a high detection rate will increase the number of false positives. Several different methods have been successfully used to achieve high detection with an acceptable number of false positives. In order to select one of these methods for our project, the following requirements were considered:

- *Hardware* - The face detector should be executed on a provided embedded system, which means limited memory bandwidth and processing power.
- *GPU* - The face detector should be implemented mainly on GPU, to unburden the CPU of the platform and to efficiently use the platform resources.
- *Real-time* - The face detector should be able to track faces and set the camera focus accordingly in real-time, in order to be useful when photographing people.

2.1 Face detection algorithms

The following sections present the most relevant methods by giving a brief description and evaluating advantages and disadvantages according to the requirements above.

2.1.1 Eigenfaces

The Eigenface method is one of the earliest successful methods for detecting, and recognizing faces. A successful implementation is described by Turk and Pentland [8]. The creation of such a detector consists of the following steps:

1. Begin with a large image set of faces.

2. Subtract the mean of all faces from every face.
3. Calculate the covariance matrix for all of the faces.
4. Calculate the eigenvalues and corresponding eigenvectors of the covariance matrix. The largest eigenvectors will be chosen as a basis for face representation. These eigenvectors are called ”eigenfaces”.

When trying to determine whether a detection window is a face, the image in the detection window is transformed to the eigenface basis. Since the eigenbasis is incomplete, image information will be lost in the transformation process. The difference, or error, between the detection window and its eigenface representation will then be thresholded to classify the detection window. If the difference is small, then there is a large probability that the detection window represents a face.

While this method has been sucessfully used, a disadvantage is that it is computationally heavy to express every detection window using the eigenface basis. This means that the eigenface method is not suitable for real-time use on an embedded system. At least not without heavy optimizations which is beyond the scope of this thesis.

2.1.2 Skin tone detection

Under normal lighting conditions, all human skin tones have certain properties in color space that can be used for skin tone detection. Faces can be found by detecting which pixels in an image represent human skin, and further examining skin regions of the image with segmentation or other classifying methods.

Thorough research on the subject has been done by Störring [7], and a working face detector using skin color detection is described by Singh, Chauhan, Vatsa and Singh [6]. They show that when all different skin tones are plotted in different color spaces, ex. RGB, YUV and HSL, they span certain volumes of the color spaces. These volumes can be enclosed by surrounding planes. Color values can then be tested to see if they lie within this skin tone volume.

The main advantage of this method related to this thesis is that it is well suited for GPU implementation. The pixel shader could efficiently examine if a pixel color lies within a skin tone volume, since this type of per pixel evaluation is what GPUs are designed for. There are however a few disadvantages with skin detection methods. For accuracy, normalized color images are required. Unless this functionality is provided by the ISP block, each image has to be normalized before use. Also, after the initial skin tone detection, the skin regions found requires further processing to determine whether they contain faces. Clustering and/or segmentation of skin regions to eliminate the effect of noise might also be required before the actual face detection. As powerful as skin tone detection is, the method can often be more useful as a preceeding step to another type of face detector, as it will reduce the number of regions that have to be tested. However, care has to be taken when processing pictures with unnatural lighting conditions so that faces are not incorrectly discarded by the skin detector.

2.1.3 Haar feature based detection

Haar feature based detection methods are based on comparing the sum of intensities in adjacent regions inside a detection window. Typically two or more

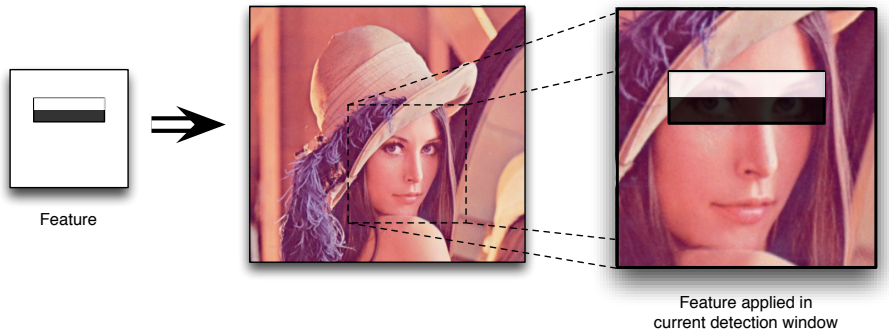


Figure 2.1: The feature on the left has a specific position and scale inside the detection window. The detection window is slided across the whole image and at each window position, the features are evaluated to see if the current detection window is centered around a face or not.

adjacent regions are combined, forming what is commonly referred to as a Haar-like feature. Features make use of the fact that objects often have some general properties. An example for faces is the darker region around the eyes compared to the forehead, as illustrated in figure 2.1. Several features are tested within a detection window to determine whether the window contains a face or not.

There are many types of basic shapes for features, as illustrated in figure 2.2. By varying the position and size of these features within the detection window, an overcomplete set of different features is constructed. The creation of the detector then consists of finding the best combination of features to separate faces from non-faces. This is commonly referred to as the training of the detector.

The main advantage of this method is that the detection process consists mainly of additions and threshold comparisons. This makes the detection fast, even on systems with limited resources, like mobile devices. On the other hand, the accuracy of the face detector is highly dependent on the database used for training. The main disadvantage is the need to calculate sums of intensities for each feature evaluation. This will require lots of lookups in the detection window, depending on the area covered by the feature. Fortunately, Viola and Jones [9] provide the integral image as a solution to this problem (see section 2.2.2). Using this technique, they successfully implemented a real-time face detector on an embedded system. The use of the integral image allows a Haar based detector to be implemented on GPU, since it is possible to calculate the sum of a region using only a few lookups. This means that this method holds all of the requirements.

2.2 Selected method: Haar feature based detection in detail

After careful evaluation of the methods above, we chose to implement Haar feature based face detection. The performance of the method is mainly dependent on the number of texture lookups and additions that the system can

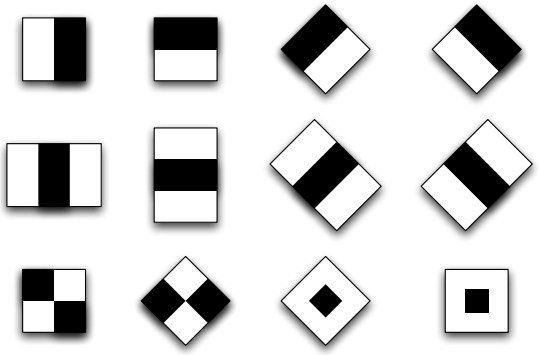


Figure 2.2: Basic types of Haar features.

handle. Graphics hardware is rapidly evolving to handle huge amounts of texture lookups and arithmetics on a per pixel basis. This makes Haar feature based detection GPGPU friendly by design.

Our main reference for the implementation of the Haar feature method was the work done by Viola and Jones [9]. They successfully implemented a face detector that runs in real-time on an embedded system, two of the requirements of this thesis project.

2.2.1 Haar-like features

The starting point for Haar feature based face detection is the set of features used. The complete set of features is constructed using basic shapes similar to those used for finding the coefficients in Haar wavelet transforms, thus the name "Haar features". This set is overcomplete and the number of features is far greater than the number of pixels in the detection window. Therefore it is not required to use all of the basic shapes to construct a feature set.

When searching for a face, a detection window is scanned with several Haar features at different scales and positions. An example with just one feature evaluation was illustrated in figure 2.1. Since feature evaluation is based on sums of intensities, the evaluation requires visiting every pixel in order to sum their intensity values. This means lots of lookups which are both time and bandwidth consuming. The concept of the integral image can be used to speed up the evaluation of features.

2.2.2 The integral image

It is possible to traverse an image once and store the intensity sums of all pixels above and to the left of the current pixel in a new image. This new image representation is the integral image. An example of how an integral image is constructed is illustrated in 2.3

An efficient way to calculate an integral image I from an original image O is to create it sequentially row-wise or column-wise from the top left corner. The following equation shows how to create an integral image (given that lookups

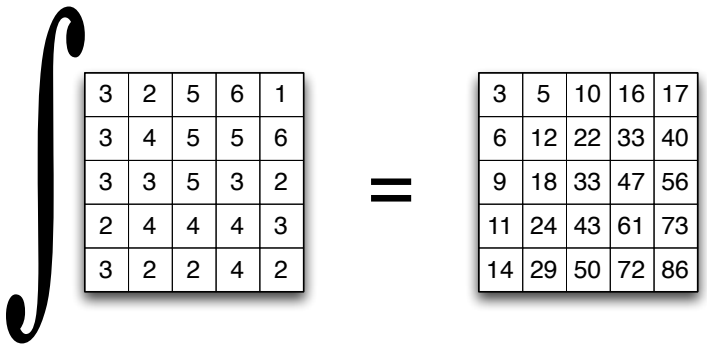


Figure 2.3: Integrating the image to the left using equation 2.1 will result in the integral image to the right.

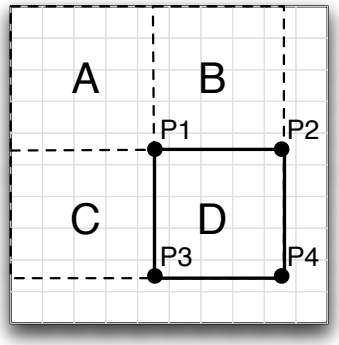


Figure 2.4: To evaluate the sum in area D, only four lookups, p_{1-4}, are required. The area can then be calculated as $D = p_1 - p_2 - p_3 + p_4$. *(Since* $p_1 - p_2 - p_3 + p_4 = A - (A + B) - (A + C) + (A + B + C + D) = D$*).*

outside the integral image are zero).

$$I_{x,y} = I_{x,y-1} + I_{x-1,y} - I_{x-1,y-1} + O_{x,y} \tag{2.1}$$

The exact same concept as the integral image has existed for a long time in the field of computer graphics, known as the summed-area table. The summed-area table is thoroughly described by Crow [3].

Since the fragment processor visits fragments at arbitrary time, the integral image, or summed-area table, can not be calculated in one pass on GPU. A working GPU implementation for calculating the summed-area table is proposed by Scheuermann [5]. But there is little to gain from implementing this on an embedded system due to the number of calculations and multiple passes needed.

One disadvantage with the integral image is that it is only possible to calculate sums of axis aligned rectangle regions, due to the way the integral image is constructed. This can easily be seen in figure 2.4. Each lookup in the integral

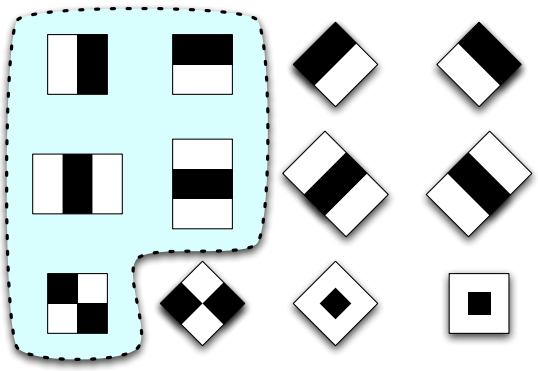


Figure 2.5: Basic types of Haar features. The highlighted ones are those used by our detector.

image contains the sum of the axis aligned rectangle from the upper left corner of the image to the lookup point. For instance the value at p_1 in the integral image in figure 2.4 contains the intensity sum of region A. Analogously p_2 contains the sum of regions $A + B$. Because of these properties of the integral image, and the real-time requirement for the thesis project, we chose to use only the axis aligned features previously proven to yield good results [4]. These features are highlighted in figure 2.5.

2.2.3 The weak classifier

Weak classifiers are constructed using one or a few Haar features with trained threshold values. In this thesis we used one feature for every weak classifer. These classify a detection window as positive (face) or negative (non-face) depending on if the feature evaluation is above or below the corresponding treshold. The result from just one weak classifier does not give enough information, but several combined into a strong classifier can determine whether a detection window contains a face or not.

2.2.4 The strong classifier

The strong classifier is a combination of several weak classifiers. Each of the weak classifiers are given weights depending on their detection accuracy. When classifying a detection window with a strong classifier, all of the weak classifiers are evaluated. The pretrained weights of the weak classifiers that classify the window as a face are added together. In the end, the sum of weights is compared with a predefined threshold to determine if the strong classifier classifies the detection window as a face or not. The threshold for the strong classifier has to be chosen to maximize the amount of correct detections while maintaining a low number of false positives. If the false positive rate is too high, the strong classifier needs more weak classifiers. The process of choosing weak classifiers and combining them is described in section 3.2.2.

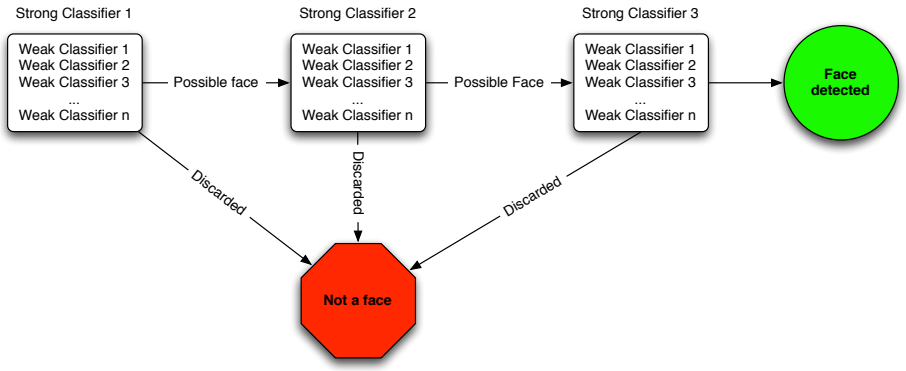


Figure 2.6: A cascade of three strong classifiers.

2.2.5 The cascade

Instead of evaluating one large strong classifier on a detection window, Viola and Jones [9] suggest the concept of a cascade. It is simply a sequence of strong classifiers which all have a high correct detection rate (ex. 0.99), and a pretty low false positive rate (ex. 0.30). If a strong classifier classifies a detection window as a non-face, it is immediately rejected (as illustrated in figure 2.6). This way, the only time the whole cascade is evaluated is when an actual face, or something similar to a face is found. Since most detection windows do not actually contain a face, the full depth of the cascade will seldom have to be evaluated. Therefore, the average amount of evaluations for each detection window will be lower than if only one strong classifier is used. The minimum possible average of weak classifier evaluations will be equal to the number of weak classifiers in the first strong classifier of the cascade, since the first step in the cascade will always be executed. Viola and Jones [9] suggest that the trained cascade should be preceded by a strong classifier with only two relevant weak classifiers with a correct detection rate of approximately 1.00. Any two weak classifiers with a false positive rate < 1.00 will help lower the average of evaluations considerably.

For each strong classifier i in a cascade, we denote the correct detection rate cd_i, and the false positive rate fp_i. Then the correct detection rate cd_{final} and false positive rate fp_{final} for the full cascade will be:

$$cd_{final} = \prod_{i=1}^{n} cd_i \qquad fp_{final} = \prod_{i=1}^{n} fp_i \tag{2.2}$$

This means that if every step of the cascade has a $cd_i = 0.99$ and $fp_i = 0.30$, a 10 step cascade will have a final correct detection rate $cd_{final} \approx 0.90$ and a final false positive rate $fp_{final} \approx 5.9e^{-6}$

Chapter 3

Training the face detector

We trained our face detector using the adaboost algoritm as described by Viola and Jones [9] and Meynet [4]. We first implemented the training framework in Mathworks Matlab, due to the mathematic and statistic nature of the method. When the results proved that our implementation was going in the right direction, we switched to C++, speeding up the training process by a factor of approximately 100.

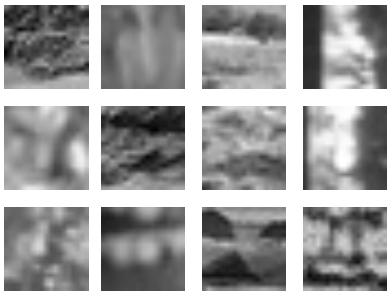
3.1 Face and non-face database

In order to train our face detector we needed a database of faces and non-faces as a statistical input. We chose to use the Cambridge AT&T Laboratories face database [2], since it was freely available and also contained a set of non-faces for the training and a test set. There were 2429 face images and 4548 non-face images in the training database. All the images were 19x19 pixels in grayscale, stored in pgm format. Using only the original AT&T database we got too high false positive rates when evaluated on the test set included. To lower the number of false positives, we expanded the non-face examples in the training set. This was done by running a primitive face detector (trained from the first training set) on images that did not contain any faces. Every detection in that set was obviously a false positive that could be collected and included as a non-face example in the training set. Examples from the Cambridge database are shown in figure 3.1(a), and examples of false positives collected by our primitive detector are shown in figure 3.1(b).

3.2 The training steps

We started by constructing a large set of Haar-like features, using the base feature types illustrated in figure 2.5. These base types can all be inverted, so we decided to combine every base feature type with a polarity, creating twice as many base feature types. Since our database consists of 19x19 pixel images, we did not create features with width or height larger than 19 pixels. To make the resulting detector more robust, we decided not to use features with a width or height less than 3 pixels. Each combination of base types, polarities, widths and heights where positioned at every possible offset within a 19x19 pixel frame

3.1(a): Example faces from the Cambridge AT&T Laboratories database.



3.1(b): Examples of difficult false positives collected using a primitive face detector by scanning images not containing faces.

Figure 3.1: Examples of images from the training set.

to create the complete set of features used. The total amount of features can be describes as:

$$F_{total} = F_{base} \cdot 2 \cdot \sum_{w=w_{min}}^{W} \sum_{h=h_{min}}^{H} ((W+1)-w)((H+1)-h) \tag{3.1}$$

Where F_{total} is the total amount of possible features, F_{base} is the amount of base features types (illustrated in figure 2.5), 2 represents that every feature has two polarities, W is the window width, H the window height, w_{min} the minimum feature width and h_{min} the minimum feature height. The number of possible offsets in equation 3.1 is dependent on the current width w and height h. The offsets for every possible width and height are summed to express the total number of size and position combinations. In our case, the total amount of features was:

$$F_{total} = 5 \cdot 2 \cdot \sum_{w=3}^{19} \sum_{h=3}^{19} ((19+1)-w)((19+1)-h) = 234090 \tag{3.2}$$

3.2.1 Weak classifier training

The 234090 features we created were evaluated one by one on the whole training set to determine the feature individual thresholds that best separates faces from non-faces, thus creating weak classifiers. To find the optimal thresholds, we used the following approach for each weak classifier:

1. Start with the lowest possible threshold.
2. Evaluate the weak classifier with the current threshold on every face example, and store the sum of correctly classified faces in a histogram H_{faces} at the current threshold.
3. Evaluate the weak classifier with the current threshold on every non-face example, and store the sum of *incorrectly classified non-faces* in another histogram $H_{non-faces}$ at the current threshold.

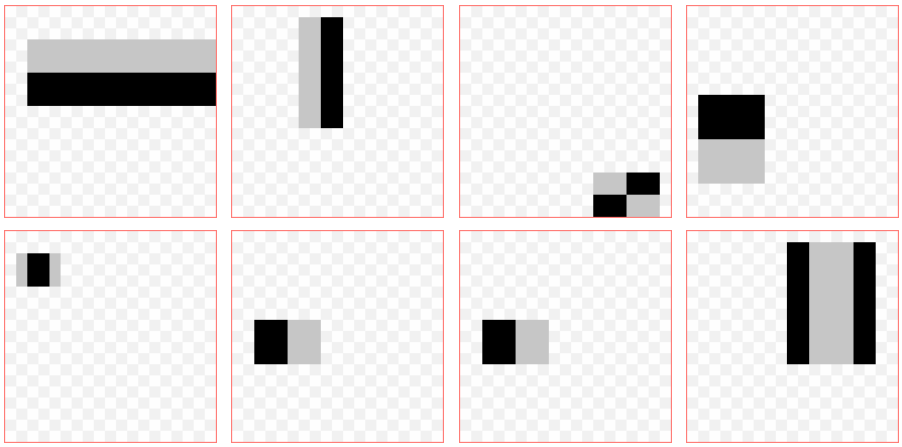


Figure 3.2: Some examples of weak classifiers used in the detector. The position, size, polarity and base feature type is illustrated by the gray and black rectangular areas. Each weak classifier also has a threshold which is not included in the illustration.

4. Increase the threshold to the next discrete value and start again at step 2 until all thresholds have been evaluated.

5. Compare H_{faces} with $H_{non-faces}$ and find the threshold t that maximizes the difference function

$$\Delta(t) = |H_{faces}(t) - H_{non-faces}(t)| \tag{3.3}$$

   This will be the threshold that best separate the faces from the non-faces.

The thresholds we found were then stored for every weak classifier. Examples of weak classifiers are illustrated in figure 3.2.

3.2.2 Strong classifier training using adaboost

In order to collect suitable weak classifiers to combine into strong classifiers, we implemented the adaboost algorithm as follows:

1. Weight all images of the training set so the sum of weights for the face set and the sum of weights for the non-face set are both 0.5. The total sum of weights will then be 1.

2. Use the weak classifiers to classify all images in the training set. Every time a weak classifier makes an incorrect classification (i.e. a face image is classified as non-face or vice versa), its error statistics is increased by the weight of that image.

3. After every weak classifier has been used to classify every image, select the one with the smallest error statistics.

4. Use the selected weak classifier to evaluate all images again, updating the weight of each image depending on whether the selected weak classifier

classifies it correctly. If an image is correctly classified its weight is increased, otherwise it is left unchanged.

5. When all weights have been updated, normalize them so that their sum is once again 1.

6. Finally, append the selected weak classifier to the strong classifier.

7. Returns to step 2 to find the next weak classifier.

When the process is repeated, the error values that are stored for every classifier will be dependent of the previously chosen weak classifiers. This way, each strong classifier contains a set of weak classifiers that are all good at detecting similar types of faces, or discarding similar types of non-faces.

The strong classifier stores a weight for every weak classifier it contains, indicating how well the individual weak classifier classified the complete set. Every time a weak classifier is appended, the strong classifier is evaluated on the complete set to determine if another weak classifier is needed. This is done by comparing the correct detection rate and false positive rate of the strong classifier to a wanted value. When the strong classifier is a part of a cascade, its correct detection rate and false positive rate are cd_i and fp_i of equation 2.2.

To evaluate a strong classifier on a detection window, its weak classifiers are evaluated one by one. The weights of the weak classifiers that classify the detection window as a face are added together. That sum of weights is then compared to a threshold to determine the result of the strong classifier. The result of the strong classifier simply indicates if enough weak classifiers classified the detection window as a face. The threshold of the strong classifier must be retrained when a weak classifier is added. We did this by adjusting the threshold until a correct detection rate of 0.99 was achieved, and added more weak classifiers until the false positive rate was acceptably low. Since we trained strong classifiers to combine into a cascade, the acceptable false positive rate of each strong classifier was as high as $0.3 - 0.5$ (as described in section 2.2.5).

3.2.3 The cascade

The training of our cascade was performed by iteratively training strong classifiers using adaboost. The first strong classifier was trained with the whole training set of faces and non-faces. The second strong classifier used a training set containing all faces but only the non-faces that the previous strong classifier could not classify correctly. In this way, strong classifiers in the later steps of the cascade can focus on different non-faces than the previous steps. This means that the cascade consists of strong classifiers that each are good at separating *different* sets of non-faces from the faces. As an optimization of the algorithm, the thresholds of the weak classifiers can be retrained after updating the training set. This will remove dependencies from discarded non-face examples. The iterative method of training the cascade is illustrated in figure 3.3. The initial strong classifiers were trained to achieve a correct detection rate of 0.99 and a false positive rate of 0.30. In later steps of the cascade, we increased the required false positive rate to approximately 0.80. Selecting the wanted false positive rate was tricky, since it is not worth adding a large amount of weak classifiers to achieve only a small decrease of the false positive rate. Even so,

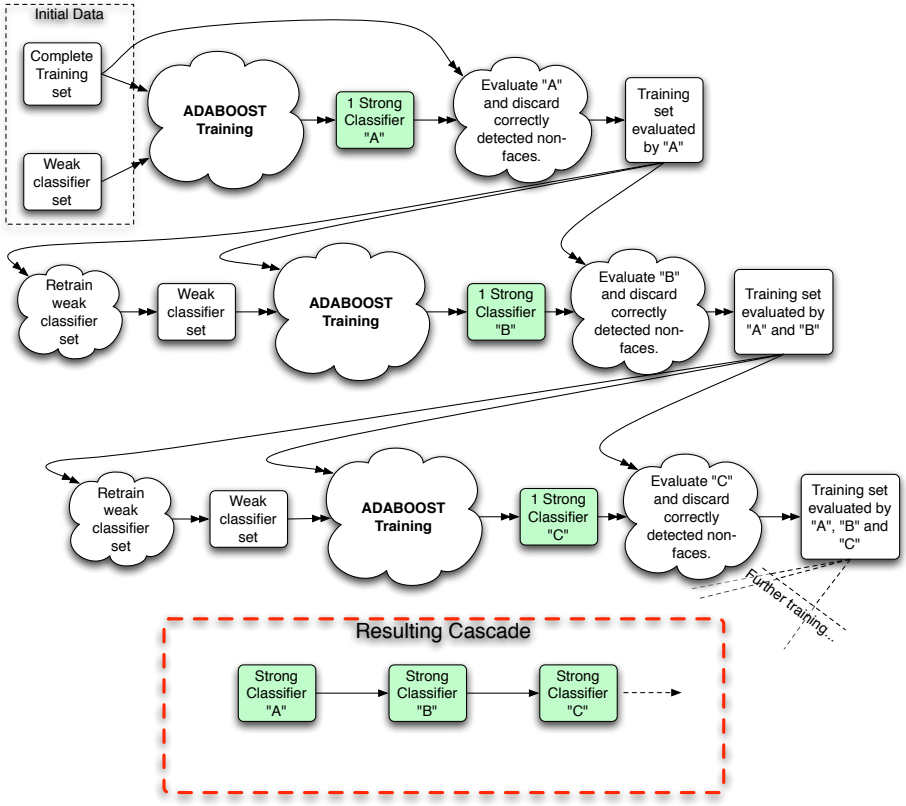


Figure 3.3: The process of training a cascade as described in section 3.2.3.

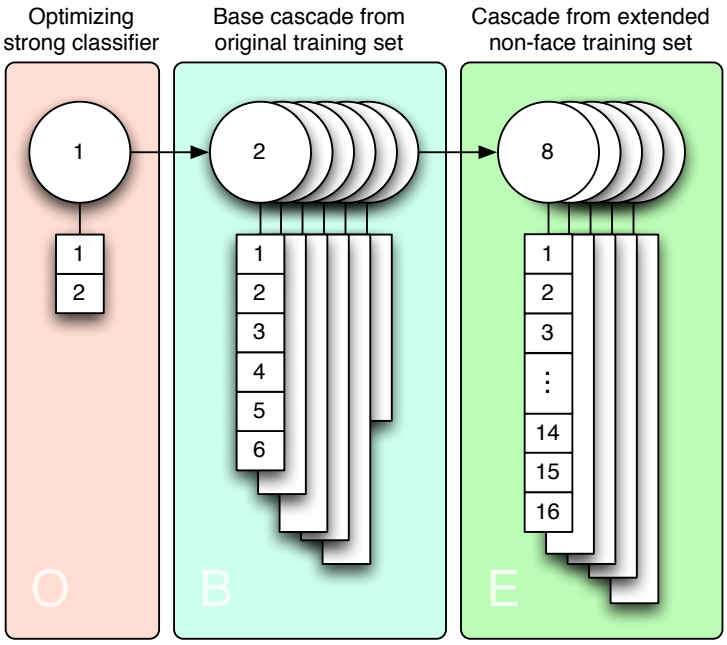


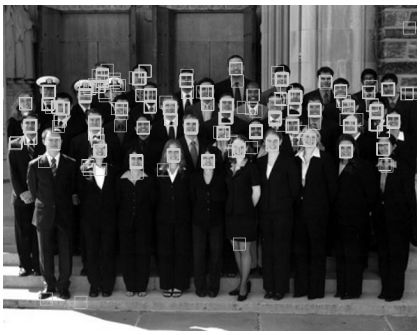
Figure 3.4: This is our final cascade, containing 12 strong classifiers. The first part is an optimization using two weak classifiers. The middle part is the cascade from the training on the original Cambridge AT&T Laboratories database. The final part of the cascade is the result of training on the extended non-face set.

with several steps in the cascade, the total false positive rate will still steadily decrease, as described in section 2.2.5. The cascade we trained from the initial database had a correct detection rate above 0.9 and a false positive rate of 0. This sounds like an optimal false positive rate, but since it is measured on the training set, it is actually not good at all. This is discussed further in section 3.3.

3.3 Training results

The cascade we obtained from training on the Cambridge AT&T Laboratories database using the algorithm described in section 3.2.3 consisted of 6 strong classifiers illustrated by the the middle part of the cascade in figure 3.4.

After the sixth step, the adaboost algorithm had successfully discarded all of the non-face examples. Thus further training was impossible without extending the database with more and harder non-faces. We used the method described in section 3.1 with pictures collected from the internet depicting landscapes, vegetation and interior decoration. Using these images we quickly generated a non-face set of approximately 80.000 non-faces. Since all of the collected non-faces were incorrectly classified as faces in the previous steps of the cascade, we continued the cascade training using only the new non-faces and the faces used to train the previous steps. This improved the false positive rate on the



3.5(a): Result of face detection using only the middle cascade from figure 3.4.

3.5(b): Result of face detection using the middle and the last cascade from figure 3.4

Figure 3.5: Detection examples illustrating a decrease in the number of false positives using different cascades. The false positive count in this case was lowered by a factor 5.

3.6(a): Result of face detection using only the middle cascade from figure 3.4.

3.6(b): Result of face detection using the middle and the last cascade from figure 3.4

Figure 3.6: Detection examples illustrating a decrease in the number of false positives using different cascades. The false positive count in this case was lowered by a factor 10.

detection examples shown in figures 3.5 and 3.6 by a factor between $5 - 10$.

As discussed in section 2.2.5, a single strong classifier was added as a first step in our cascade to lower the average number of features evaluated. This strong classifier was constructed using the two individually best weak classifier from the base types selected for our detector.

Statistics for different combinations of the three steps in the cascade from figure 3.4 are presented in tables 3.1, 3.2 and 3.3. The presence of the first, middle and last steps are denoted O for optimization, B for base cascade, and E for extension cascade. Column *Correct det.* contains the correct detection rate for the cascade combinations, column *False pos.* contains the false positive rate of the cascade combinations and column *Average eval.* contains the average amount of weak classifier evaluations required to reject the non-faces in the test set. It is important to note that the number of faces in the sets used to generate these metrics are much higher than the number of faces found in most pictures.

Therefore the number of average weak classifier evaluations is *not* calculated using the complete set, but rather using only the non-face examples.

Table 3.1: Different cascade combinations evaluated on the original training set in the Cambridge AT&T Laboratories database

Cascade			Correct det.	False pos.	Average eval.
O	-	-	0.9716	0.3467	2.0000
-	B	-	0.9671	0	11.0330
-	-	E	0.9737	0.0836	28.3967
-	B	E	0.9481	0	11.0330
O	B	E	0.9329	0	8.5871

The results in table 3.1 show that cascade B has a false positive rate of 0 on the training set as discussed in the beginning of this section. Another noteworthy result is the optimization cascade, which alone was able to reject almost two thirds of the non-faces.

Table 3.2: Different cascade combinations evaluated on the collected false positives and the original faces from the Cambridge AT&T Laboratories database

Cascade			Correct det.	False pos.	Average eval.
O	-	-	0.9716	0.7910	2.0000
-	B	-	0.9671	1.0000	81.0000
-	-	E	0.9737	0.1558	38.3886
-	B	E	0.9481	0.1557	119.3886
O	B	E	0.9329	0.1467	99.2246

In table 3.2, the B cascade has the same correct detection rate as in table 3.1 since both sets contain the same faces. Its false positive rate however, is 1. This is due to the fact that all non-face examples in that set are false positives collected using the B cascade. This is also the explanation of the fact that the average number of weak classifier evaluations is equal to the total amount weak classifiers in cascade B. The high average amount of evaluations for all of the cascade combinations in table 3.2 shows that the non-face examples collected were much harder to classify than those supplied with the Cambridge AT&T Laboratories database training set.

Table 3.3: Different cascade combinations evaluated on the original test set in the Cambridge AT&T Laboratories database

Cascade			Correct det.	False pos.	Average eval.
O	-	-	0.7500	0.3397	2.0000
-	B	-	0.1822	0.0083	10.9367
-	-	E	0.3157	0.0313	21.6856
-	B	E	0.1314	0.0034	11.4712
O	B	E	0.1292	0.0034	8.7759

The results on the test set in the Cambridge AT&T Laboratories database (table 3.3) were poor compared to the same measures on the sets used for

training (table 3.1 and 3.2). This shows the importance of selecting a suitable training set when creating a face detector. A trained detector will be limited to find faces similar to those in its training set.

As can be seen in the tables above, the use of the optimization cascade reduces not only the false positive rate, but also the correct detection rate as described in equation 2.2. However, the average amount of evaluations is always lower when using the optimization cascade. Therefore, the optimization stage is useful when higher performance is wanted.

Chapter 4

Implementation of the face detector

Implementing a face detector framework is challenging as many different subsystems can be used. Obviously, every face detector needs an image source. This could simply be an image file, but the image provider could also be a camera, like in this thesis project. After the detection process, the results can be used for whatever purpose the face detector was implemented. It is often useful to visualize the result of the detection on screen, or in an image. This representation will require some type of post-processing, like marking the faces found. The different parts in our detection framework are illustrated in figure 4.1.

Since one of the requirements for our thesis project was to use the GPU for the detection process, our framework includes a GPU part in addition to the classic components. This chapter will focus on the implementation of the detector part of the framework.

4.1 GPU implementation

As described in chapter 2, one of the requirements was that the face detector should be implemented mainly on GPU, to unburden the CPU of the platform and to efficiently use the platform resources. Early tests on the target platform showed that it could calculate approximately 75 integral images of size 640*480 pixels per second using the CPU. Due to these results, and the fact that a GPU implementation of the integral image construction would require several passes of the fragment shader, we decided to create the integral image on CPU and leave only the actual evaluation of features to the GPU. This will conform to the second part of the requirement: to efficiently use the platform resources.

Unfortunately for the handling of the integral image, OpenGL ES 2.0 does not specify a high precision texture format suitable for representing the integral image. Instead of simply accessing the values of the integral image in the shader program using a regular sampler, we had to do it in a more resource demanding way.

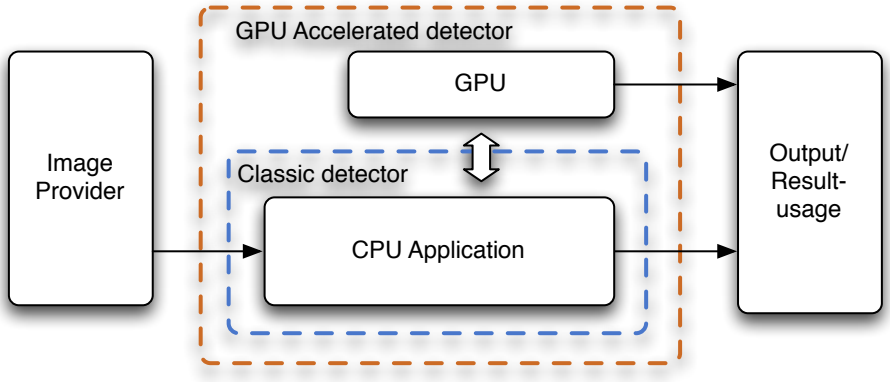


Figure 4.1: A framework for face detection with and without GPU acceleration.

4.1.1 Accessing the integral image on GPU

The integral image needs to store the intensity sum of all pixels in the original image, therefore it needs a high precision storage type. The size of the storage type has to be able to store values larger or equal to $w \cdot h \cdot I$ where w is the image width, h is the height of the image and I is the number of possible intensity values. If the original image has 256 levels of intensity and the integral image storage type is an array of 32 bit `unsigned int` values, the following simple calculation shows how large the original image can be:

$$w \cdot h \cdot 2^8 \leq 2^{32} \Rightarrow w \cdot h \leq 2^{24}$$

Given the values above, the image is limited to a size of 2^{24} pixels (approximately 16 megapixels), which is more than enough to detect faces on a 19x19 pixel scale. In GLSL ES 2.0 the `highp` precision qualifier supplies sufficient precision for handling an integral image. The GLSL ES 2.0 standard defines the `highp` precision qualifier in the fragment shader as optional, but fortunately our hardware has a unified shader GPU and therefore supports the `highp` qualifier in both vertex and fragment shader programs.

It is possible to upload the 32 bit integral image to the GPU by configuring a texture to contain RGBA values with storage type `unsigned byte`. After uploading the data to the GPU, the 32 bit `unsigned int` values will be treated as four `unsigned byte` values. When accessing the texture on the GPU, the values are read individually as `float` values that vary in 2^8 steps between 0 and 1 (illustrated in figure 4.2). Therefore we first have to scale them to vary in 2^8 steps between 0 and 255. Then we would like to shift them bitwise to their correct magnitude, unfortunately bitwise operations are not supported in GLSL ES 2.0. So we have to use multiplication instead. The conversion is illustrated in figure 4.3.

4.1.2 GPU implementation of the detector

When the integral image can be accessed and converted back to its actual value in the fragment program, it is possible to implement the cascade in shader code.

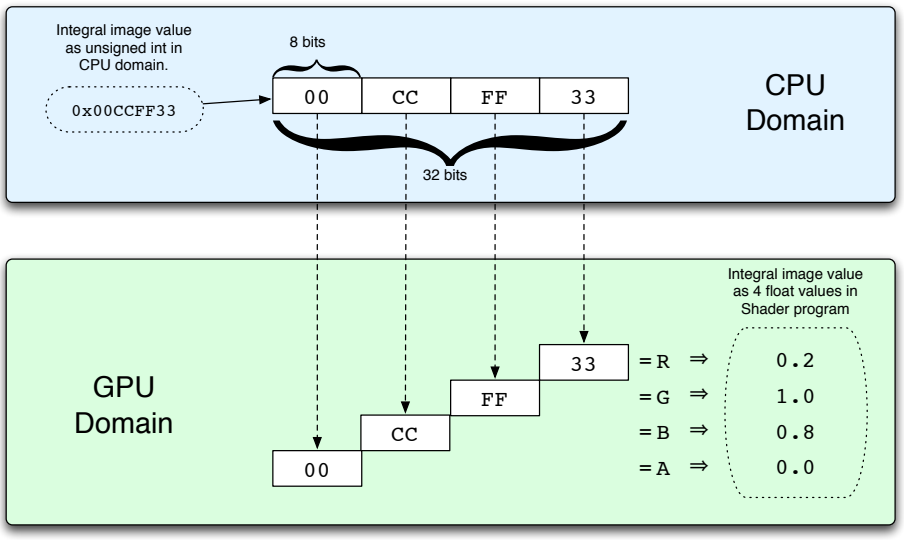


Figure 4.2: How the unsigned integer value is represented when used in the shader program.

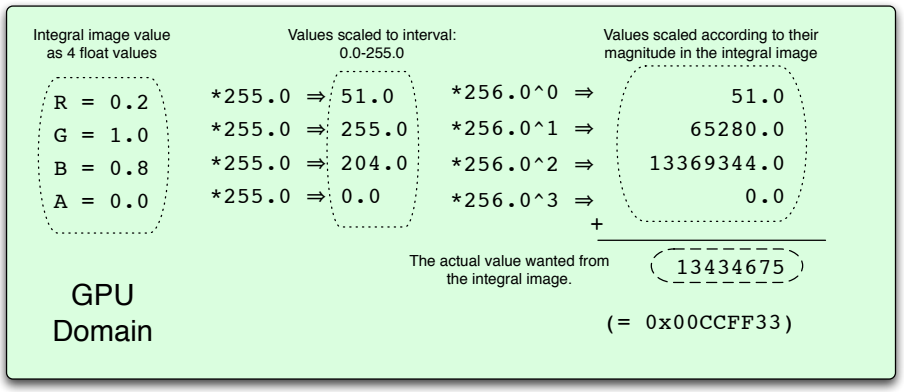


Figure 4.3: How the texture lookup float values are converted into the integral image value in the shader program.

The code for our GPU version of the detector is abstracted into weak classifier evaluation, strong classifier evaluation and cascade evaluation.

Weak classifier implementation

To evaluate the weak Haar classifiers in the strong classifiers, we created functions in GLSL ES code according to this prototype:

```
uniform highp float size;
float evalHaarClassifierTypeA(    xpos,       ypos,     width,
                height, polarity, threshold)
{
    // Evaluate classifier in current detection window.
    // Return 1.0 if detection window classified as
    // a face, otherwise 0.0.
}
```

The process of evaluating a weak classifier is as follows:

1. Modify the window-size relative input parameters to fit the current detection window size, indicated by the uniform `size`.
2. Do lookups in the texture containing the integral image.
3. Convert the values from step 2 to the actual values of the integral image, as described in figure 4.3.
4. Subtract and add integral image values according to classifier type scheme.
5. Compare the result value with the given threshold, using the polarity parameter to indicate whether the result value should be larger or smaller than the threshold.
6. Return result of comparison (1.0 for true, 0.0 for false).

These functions are created individually for the different types of haar classifiers illustrated in figure 2.5.

Strong classifier implementation

The only difference between different strong classifiers is the number of weak classifiers that are evaluated and the properties of these weak classifiers (position, size, type, polarity and threshold). After the training process is complete, it is a simple matter to store the strong classifiers as shader code functions. The properties of the weak classifiers can be stored as static values in function calls, and all weights and thresholds can be stored in variable assignments as the following code prototype shows:

```
bool strong1()
{
    float r1    =    1.47299*evalHaarClassifierTypeA( 8.0, 5.0,    5.0,
                                                      19.0, 1.0, -611.0);
    float r2    =    1.27765*evalHaarClassifierTypeB(20.0, 0.0,    4.0,
                                                      23.0,-1.0,  137.0);
    float r3    =    1.30542*evalHaarClassifierTypeA( 8.0, 3.0,   10.0,
                                                      11.0, 1.0,-1718.0);
    float thresh    =    2.02803;
    float res    =    r1 + r2 + r3;
    return res>thresh;
}
```

The variables `r1`, `r2` and `r3` will contain the corresponding weak classifiers weights if the weak classifiers classified the detection window as a face, otherwise the value will be 0. When determining the result of the strong classifier, the sum is simply compared to the strong classifier threshold.

Cascade implementation

In our implementation of the GPU cascade, we set the color of the pixel to be white if a face is detected and black otherwise. The actual cascade evaluation code was straightforward:

```
void main
{
    gl_FragColor = vec4(0.0,0.0,0.0,0.0);
    if (!strong1()){
        discard;
    }
    if (!strong2()){
        discard;
    }
    .
    .
    .
    if (!strongN()){
        discard;
    }
    gl_FragColor = vec4(1.0,1.0,1.0,1.0);
}
```

There were some issues when implementing the cascade on the GPU. Shader programs can only have a limited size, defined by the OpenGL ES 2.0 conformance tests (which are not finished at the time of writing this thesis). Because of the growing number of weak classifiers in each strong classifier, the cascade implementation resulted in long shader programs. The OpenGL ES 2.0 specification requires that compiling too long shader programs result in a compilation error. The drivers supplied with the platform refused to compile shader code containing the full cascade, but did not explicitly indicate that the length of the code was the problem. When the cascade length was reduced to only one strong classifier, the code compiled successfully. Since the same function calls were used in both cases, only fewer times in the latter case, we deduced that the actual problem was the length of the code. The solution was to split the cascade into several passes. This was implemented using frame buffer objects

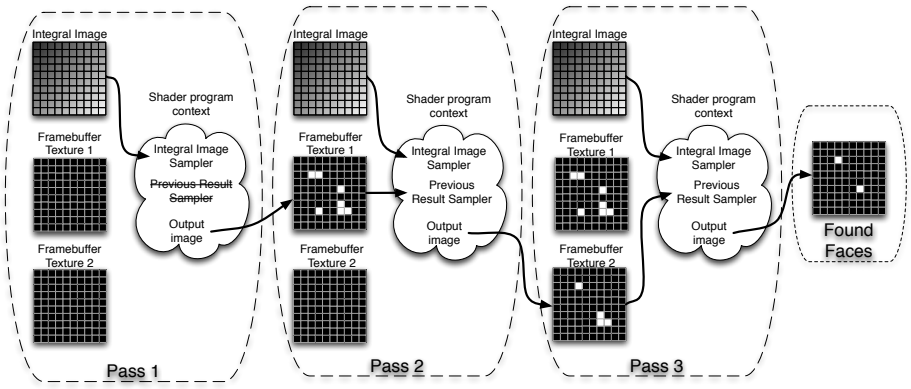


Figure 4.4: The cascade implemented using several passes on the fragment shader.

which all share the same integral image texture, and every pass uses the result from the previous step. This method is illustrated in figure 4.4, but when trying implement the method on the platform, we encountered driver related issues.

4.1.3 GPU results and issues

Before the delivery of the graphics drivers for the platform, a simulated OpenGL ES 2.0 environment running on a desktop PC with Linux was used. In the simulated environment, the OpenGL ES 2.0 calls mapped to the desktop OpenGL 2.x hardware. Since OpenGL ES 2.0 is not a true subset of OpenGL 2.x, some of the functionality in OpenGL ES 2.0, like high precision `float` values, had no counterpart in the simulated environment. The `highp` precision qualifier was needed to convert the integral image data (as described in section 4.1.1), but since there is no precision qualifier support in OpenGL 2.x, the qualifiers were simply ignored. This was solved by disabling the hardware support for OpenGL 2.x vertex and fragment shaders. By forcing the shaders to run in software mode, the high precision was emulated correctly. This decreased the performance, but we could at least verify that the face detector behaved as expected when implemented in GLSL ES 2.0.

Changing from the simulated environment to the actual hardware proved more troublesome than anticipated. The `highp` precision qualifier did not seem to work unless a redundant calculation using `highp int` values was evaluated between our texture lookup calls and the conversion to integral image values. Since the functionality did not depend on the result of the redundant calculation, it was abstracted into a function `void precisionfix(void)`. This seemingly pointless function was then called inside every weak classifier evaluation. This expanded the shader program length, further restricting the cascade implementation. To the end of this thesis, we did not find the source of this noteworthy behaviour. Due to these circumstances and the late delivery of the GPU drivers, we could not successfully implement the detector with GPU acceleration. Instead we moved on to a CPU reference implementation of the detector to be able to explore the focus control on the target platform.

Figure 4.5: Output after evaluating our final cascade in our CPU reference implementation. A rectangle overlap test algorithm have grouped multiple hits together.

4.2 CPU reference implementation

Due to the restrictions in our supplied GPU drivers, a reference implementation in C running completely on the platform CPU was implemented. This implementation is basically two for-loops that evaluate the cascade in every detection window of the resulting image, which would represent the fragment shader executing the cascade on every fragment on the GPU. With this implementation, we were able to execute the full cascade with promising results. The detector fetches images from the camera module of the platform, detects faces and presents the results on the screen of the platform. The complete framework is further described in chapter 5.

4.3 Example detections

Example outputs from our CPU reference implementation are shown in figure 4.5, 4.6 and 4.7. These were all created by running the detector on the supplied platform.

Figure 4.6: Output after evaluating our final cascade in the CPU reference implementation. A rectangle overlap test algorithm have grouped multiple hits together.

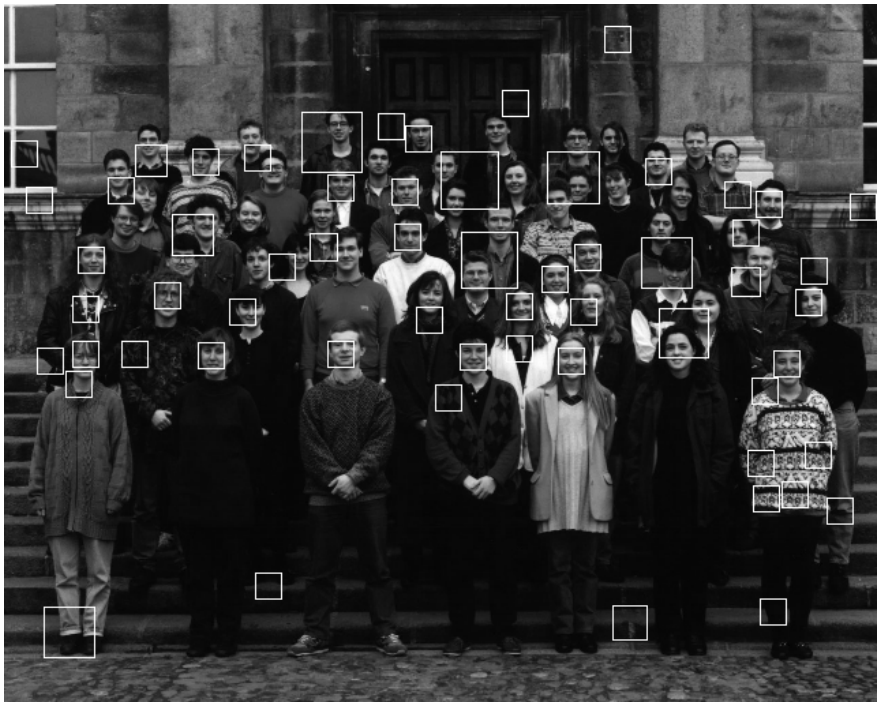


Figure 4.7: Output after evaluating our final cascade in our CPU reference implementation. A rectangle overlap test algorithm have grouped multiple hits together.

Chapter 5

Detection on the target platform

The target platform for this master thesis project had a camera module with focus capabilities and a QVGA touch screen. The platform environment was embedded Linux, with V4L2 drivers for the camera, and DirectFB support for handling input events and showing images on the screen.

5.1 Fetching and showing images

In addition to the V4L2 API, the camera included proprietary drivers supporting ISP functionality like exposure and white balance. The camera was completely controlled through these drivers. In the detection framework, the camera was set up to supply images in pixel format YUV422. This format consists of a 4 bit Y (luminance) channel, 2 bit U (blue chrominance) channel and a 2 bit V (red chrominance) channel for every pixel. Since the detection method used does not need high resolution images, a resolution fitting the screen was selected.

DirectFB has support for the YUV422 pixel format, therefore the images were kept in YUV422 for the final on-screen representation. Since the detector only uses the intensity values in the image, the color information was discarded before creating the integral image.

To display images on screen, a pointer to the screen buffer was requested from the DirectFB context. This pointer could then be used to write images to screen. Images were fetched from the camera module by requesting a pointer to the memory area where the camera module had stored the image. This setup is illustrated in figure 5.1. From the camera image, we created an integral image used to evaluate the cascade. The resulting detections were indicated by drawing squares on top of the fetched camera image, and copying the result image to the screen buffer memory area. This application flow is illustrated in figure 5.2.

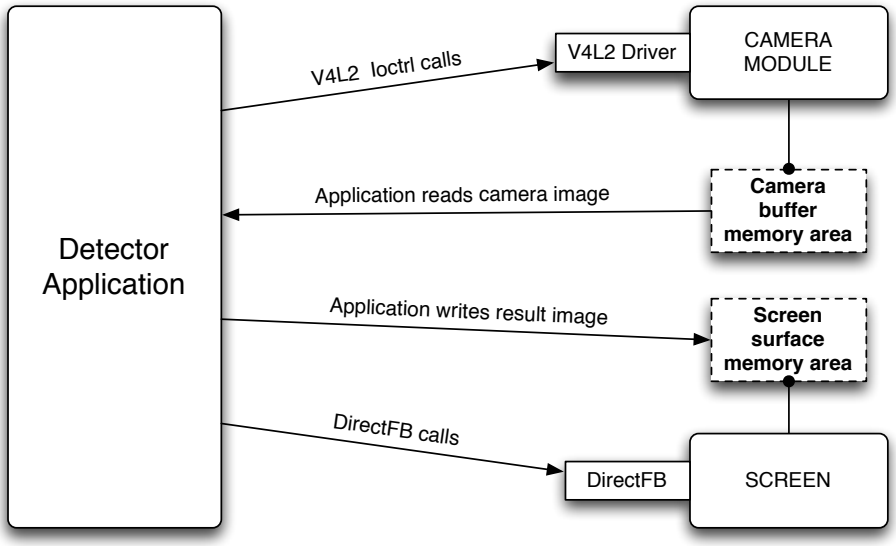


Figure 5.1: The setup of the target platforms camera and screen.

5.2 Controlling the focus

Faces are often the subject of interest when taking photographs. Face detection can therefore be used to select a region in the viewfinder for camera focus setup.

5.2.1 Focus control using GPU

An implementation of region based autofocus control using GPU is suggested by Berglund and Andreen [1]. When a region of an image is detected as a face, the contrast of adjacent pixels in that region can be measured while the focus is changing. If the contrast increases, the region becomes more focused. When a peak occurs, the wanted focal length of the camera has been found. If different parts of the measured region have different distances to the camera, false peaks can occur - in the sense that something else than the wanted area is in focus. Using face detection, regions containing only faces can be used to find the optimal focal length.

5.2.2 Implementation issues

Platform drivers were supplied that could set the focus to values between $0-100$, mapping to the complete focal span of the camera. Unfortunately the drivers did not actually set the focus, thus preventing the implementation of focus control using face detection techniques. At the time of writing this report, the supplier has not yet provided a working focus control.

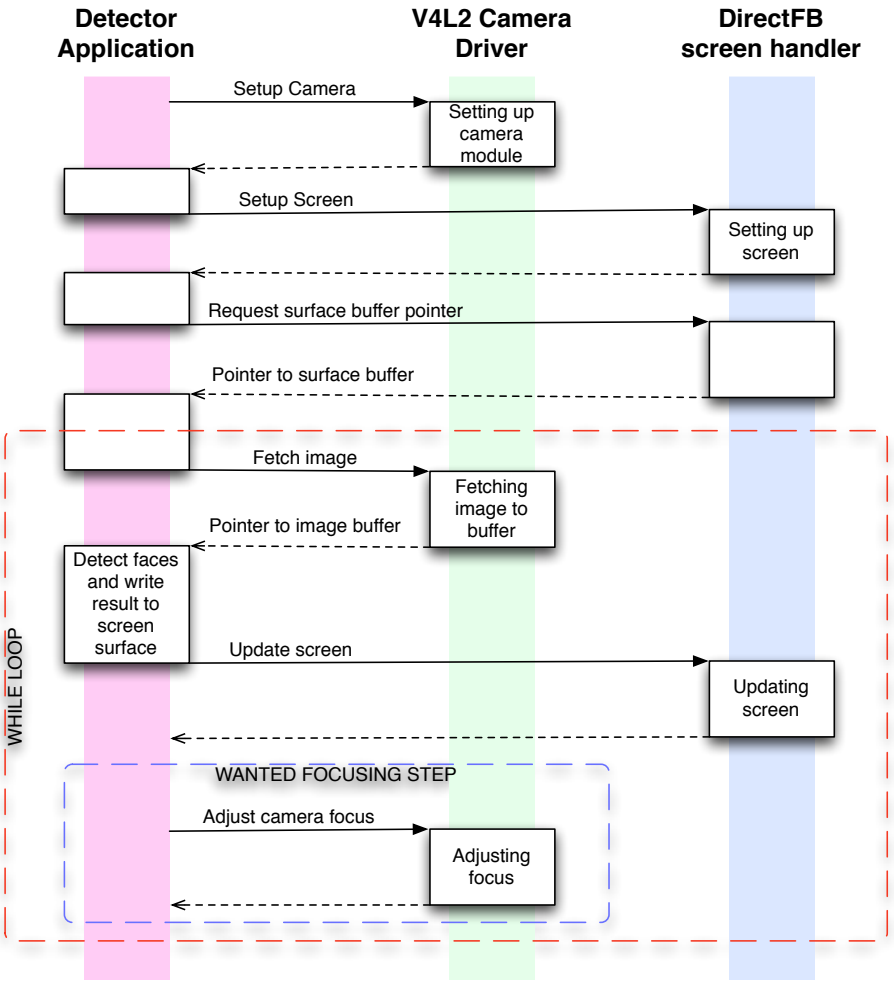


Figure 5.2: The process of fetching images, detecting faces in them and presenting them on screen. The step of setting the camera focus could not be implemented with the given drivers of the platform.

5.3 Performance

Because of the issues described in section 4.1.3, we were unable to benchmark the performance of a GPU implementation. And since the running concept in the simulated environment uses a software shader driver, the execution time in this environment is irrelevant.

The CPU reference implementation was compiled with gcc, using an ARM toolchain with optimization level 3 (the -O3 flag). When the cascade evaluation is switched off, the framework fetches 320 times 240 pixel images, integrates them and writes a result image to the screen at a frame rate of approximately 4 frames per second.

The detector was configured to evaluate the cascade in detection windows, slided across the image in steps of $1 - 10$ pixels, depending on the size of the detection window. For detection windows of size 19 times 19 pixels, the step size is 1 pixel, and for detection windows of size 190 times 190 pixels, the step size is 10 pixels. When the detector is running with detection windows of 4 different sizes between 60 and 100 pixels (in both width and height), it can detect and follow a face at a frame rate of 2 frames per second. When the detector is running with detection windows of 4 different sizes between 40 and 100 pixels, it can detect and follow a face at a frame rate of 1 frame per second.

5.4 Contrast issues

The images fetched from the camera module had poor contrast, making face detection difficult. Normalizing the images before detection was therefore necessary. The camera module had support for histogram equalization, but we were unable to set up the histogram module using the supplied drivers. Therefore we implemented a simple software histogram equalization to obtain higher contrast images for the detector.

Chapter 6

Conclusions and future work

6.1 Conclusion

In the beginning of the thesis, we evaluated different face detection methods according to the given requirements. We selected the method thought to be most suitable and implemented a training framework for such a detector. The adaboost training proved to be challenging in a number of ways, and was constantly refined throughout the duration of the thesis project. Using the results from the training, a detector prototype was implemented in a simulated OpenGL ES 2.0 environment. A framework for the target platform - capabable of fetching images from the camera module, processing them and presenting them on the platform screen - was successfully implemented. Using a CPU reference implementation of the detector, this framework ran at a frame rate of approximately $1 - 2$ frames per second without platform-specific optimization.

The initial objective of this thesis was to implement a face detection algorithm on GPU using OpenGL ES 2.0. When moving from the simulated environment to the actual target hardware, the graphics framework of the target was pushed to its limits. This was manifested in undefined behaviours, as have been previously discussed in section 4.1.3. Conclusions drawn from this is that the platform with the current drivers might be ready for regular 3D graphics development, but not yet for GPGPU.

Because of this, we were unable to gather sufficient results to determine whether a GPGPU accelerated face detection was implementable. Indications from our incomplete GPU implementation suggest that a complete detector, using the selected method, would have been too slow for real-time use.

6.2 Discussion and future work

6.2.1 Open GL ES 2.0

Writing a program in a simulated OpenGL ES 2.0 environment is not enough, it is always important to test with actual target drivers. Since OpenGL ES 2.0 is not a true subset of OpenGL 2.0, an application working in a simulated environment could be problematic to run on actual ES hardware. A lot of time and effort can be saved if platform specific bugs are found in early stages of

development.

6.2.2 Normalized detection windows

For the detector to be illumination invariant, it must first be trained on a normalized database. Every single detection window must then be individually normalized during detection. Viola and Jones [9] suggest that the detection window normalization should be done by adjusting the threshold values of the weak classifiers, instead of normalizing the detection window on a per pixel basis. The intensity variance σ^2 in each detection window could be used for the threshold adjustment. The variance could be calculated using another type of integral image containing the sum of squared intensity values from the original image. Such a representation would require even higher precision. Therefore, this method of normalization was not feasible as a GPU implementation on the target platform.

Our detector however, was trained with a database that was not normalized. Therefore it can detect faces without having to normalize every detection window. This is not as robust as the method above but will only require a normalization of the complete image, which can be done quite efficiently on CPU or by a camera ISP block.

6.2.3 Focusing algorithm optimization

When a face has been detected in an image, the size of the face will be known. The size of the face, together with biometric statistics can be used as a rough estimation of the distance from the camera to the face. This distance can be used to limit the search range of the focusing algorithm. If a detected face covers a large area of the camera's viewfinder, the focusing algorithm can predict that it is highly probable that the face is close to the camera. Similarly, if a face covers a small portion of the viewfinder, the face is probably further away.

It is important to note that this optimization would require the detection to be an actual face and not a billboard or a small photograph depicting a face.

6.2.4 Float conversion

The conversion procedure concerning `float` values described in section 4.1.1 requires a lot of operations. This is a computationally expensive way to transfer high precision values between the CPU and GPU domains. Since graphics hardware manufacturers are aware of the demands from GPGPU programmers, better support for high precision textures is expected. Desktop GPUs already support this type of as extensions. It is highly likely that also OpenGL ES hardware will evolve to include such support.

6.2.5 Choosing a suitable database

Haar feature based detection is highly dependent on the database used for training, as explained in section 3.3. Face databases are often constructed with photos taken under similar lighting conditions, using the same background in every photo. A robust face detector should not be dependent on circumstances such as background and light conditions. To create a robust detector several different

databases can be merged into one training set. Since adaboost training requires images to be consistent in size and position, the merge of different databases can be problematic. It may require a considerable amount of manual labour, like scaling and aligning the images. Some of the most successful implementations of Haar based detectors, like the one by Viola and Jones [9], have been trained with images collected by crawling the internet for random faces. This will obviously yield a higher statistic variance for the collected images compared to a database of systematically photographed faces. Higher statistic variance in the training database will have a positive impact on the detector.

Another important aspect of the database is how much of the face is cropped away. The database used in this thesis only contained the center part of the face, cropping away the outlines like hair, ears, and chin. This can be seen in figure 3.1(a). The tradeoff from cropping is that face shaped objects are easier to discard, as nothing from the outline of the face is used for the detection. It will however, be more difficult to find enough information to eliminate false positives. Which is one of the major weaknesses of our detector. In retrospect, we would have liked to use a database with faces less cropped. Unfortunately, we were unable to accquire such a database during this thesis project.

Non-face examples are also required, but a lot easier to aquire. Cutting out random parts from images not containing faces will quickly give a large non-face set. However only a fraction of these non-faces will be of any real use for the detector. Most non-faces will be easy to separate from the set of faces and thus will be discarded early. An effective way to extract more challenging non-faces is to run a primitive incomplete face detector on images not containing faces, and collect the false positives. This will give a higher difficulty amongst the non-faces that would be hard to achieve using a truly random collection method.

6.2.6 Skin tone methods

To decrease the average of weak classifier evaluations in the cascade, it is possible to do a preceding skin tone test. This would mean that the original color image has to be uploaded to the GPU, in addition to the integral image.

Before the cascade evaluations begin, a few texture lookups in the original image can be used to determine if the current search window contains enough skin color. If not, the cascade does not need to evaluate any classifiers. The smallest possible amount of texture lookups to evaluate a weak classifier is 6, and the first strong classifier in the cascade should contain at least two weak classifiers. Therefore, a skin test that rejects large areas of an image that does not contain any faces should decrease the average number of evaluated weak classifiers per pixel considerably. One problem though, is that a skin tone detection might incorrectly reject face regions due to unnatural lighting conditions. This means that a color normalization is required for robust skin tone detection. The performance benefit of this method on GPU is dependent on the branching capabilities of the hardware, as described in section 2.2.5.

Many of the difficulties encountered in this thesis were due to the need of high precision. Skin tone based methods does not require high precision values, but rather regular color values. Since GPUs are designed for handling such color values, it would be interesting to further explore the possibilities of GPU accelerated skin tone based face detection.

Chapter 7

Acknowledgements

Thanks to Beatriz Grafulla for support throughout the thesis and while writing the report.

Thanks to Harald Gustafsson for discussions about the target platform.

Thanks to Per Persson for linux support and setting up the development environment.

Thanks to Jon Hasselgren for OpenGL related discussions.

Thanks to Jim Rasmusson for some Very good pieces of advice.

Thanks to the entire Ericsson new technology and research department for supplying a friendly working environment for thesis workers.

Bibliography

[1] Fredrik Berglund and Mikael Andreen. Camera isp applied on a programmable gpu, 2008.

[2] AT&T Laboratories Cambridge and Cambridge University Computer Laboratory. http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html.

[3] Franklin C. Crow. Summed-area tables for texture mapping. http://www.cse.ucsc.edu/classes/cmps160/Fall05/papers/p207-crow.pdf, 1984.

[4] Julien Meynet. Fast face detection using adaboost. http://people.epfl.ch/julien.meynet, 2003.

[5] Thorsten Scheuermann. Summed-area tables and their application to dynamic glossy environment reflections. http://ati.amd.com/developer/gdc/GDC2005_SATEnvironmentReflections.pdf, 2005.

[6] Sanjay Kr. Singh, D. S. Chauhan, Mayank Vatsa, and Richa Singh. A robust skin color based face detection algorithm. http://www2.tku.edu.tw/~tkjse/6-4/6-4-6.pdf, 2003.

[7] Moritz Störring. Computer vision and human skin colour. http://www.cvmt.dk/~mst/Publications/phd/index.html.

[8] M. Turk and A. Pentland. Face recognition using eigenfaces. http://people.epfl.ch/julien.meynet, 1991.

[9] Paul Viola and Michael J. Jones. Rapid object detection using a boosted cascade of simple features. www.hpl.hp.com/techreports/Compaq-DEC/CRL-2001-1.pdf, 2001.